Safety Not Guaranteed:
sun.misc.Unsafe and the quest for safe alternatives

Paul Sandoz
Oracle
The “Unsafe” situation

Paul Sandoz
Oracle
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
sun.misc.Unsafe

- Internal class within the JDK
- Used inside and unfortunately outside of the JDK
  - Never intended for outside use!
- Used by many popular libraries/applications
The situation

• We would like to stop outside use of Unsafe

• Undermines the safety guarantees offered by Java

• Project Jigsaw will enforce this
Unsafe vs. Java culture

• Java developers rely on safety
  • https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm

• Understatement that this is important

• You are not supposed to worry about SEGV or memory corruption!

• Safety enables an ecosystem of libraries
Is this code safe?

```java
src/java/org/apache/cassandra/utils/FastByteComparisons.java

int minLength = Math.min(length1, length2);
int minWords = minLength / Longs.BYTES;
int offset1Adj = offset1 + BYTE_ARRAY_BASE_OFFSET;
int offset2Adj = offset2 + BYTE_ARRAY_BASE_OFFSET;

/*
 * Compare 8 bytes at a time. Benchmarking shows comparing 8 bytes at a
 * time is no slower than comparing 4 bytes at a time even on 32-bit.
 * On the other hand, it is substantially faster on 64-bit.
 */
for (int i = 0; i < minWords * Longs.BYTES; i += Longs.BYTES) {
    long lw = theUnsafe.getLong(buffer1, offset1Adj + (long) i);
    long rw = theUnsafe.getLong(buffer2, offset2Adj + (long) i);
    long diff = lw ^ rw;
```
Answer

• It is now but was not before Feb 2014
• Can cause JVM crash on Solaris/Sparc
• Bug has been present for over 2 years
• Introduced in 1.1.3 fixed in 2.0.5
• Bug: CASSANDRA-6628
• Code diff
What about this?


```java
public long getLong(int index) {
    checkIndexLength(index, SIZE_OF_LONG);
    return unsafe.getLong(base, address + index);
}
```
Answer

• No, does not account for alignment and native byte order

• Pull request is ~7 months old

I think we should just say "Slice only supports little endian machines"
What about this?

http://hg.openjdk.java.net/jdk9/dev/jdk/file/tip/src/share/classes/java/util/concurrent/atomic/AtomicLong.java

```java
public class AtomicLong extends Number implements java.io.Serializable {
    ...
    private volatile long value;
    ...
    public final void set(long newValue) {
        value = newValue;
    }
    ...
    public final boolean compareAndSet(long expect, long update) {
        return unsafe.compareAndSwapLong(this, valueOffset, expect, update);
    }
    ...
}
```
Answer

• Yes, on all mainline OpenJDK platforms
• No, on a platform not supporting compare-and-set of 64 bit values
• Issue JDK-8044616
• Compare-and-set requires a lock
• Cannot enforce atomicity w.r.t direct stores
Unsafe is a sharp tool

• Even advanced developers can get things wrong or assumptions change
• Perhaps some developers don’t care
• Many other pitfalls
• Especially security and serialization related
Developer distinctions

- 99% of Java developers don’t use, and have probably never heard of, and have no need to use, Unsafe
- 1% of Java developers use Unsafe
- Often write widely-used libraries
- Therefore much of that 99% transitively use Unsafe too without knowing it
The 1% mix

• Developers who think they need to use it but probably don’t really need to

• Advanced developers with a genuine need and don’t care about cross platform safety

• Advanced developers with a genuine need and align with Java’s culture

• Developers working on the JDK itself
Poke the bear

http://en.wikipedia.org/wiki/Wikipedia:Don't_poke_the_bear#mediaviewer/File:Male_kodiak_bear_face.JPG
Poke!
Focus on

• Developers who think they need to use it but probably don’t
• Advanced developers with a genuine need and don’t care about cross-platform safety
• Advanced developers with a genuine need and align with Java’s culture
• Developers working on the JDK itself
Advanced Unsafe Developers

• Pushing the boundaries of where and how the JVM is used
• Often vocal & innovative
• Develop popular libraries
• Good for the Java community
The Unsafe survey

<table>
<thead>
<tr>
<th>6. Do you have an optional dependency on Unsafe to ensure code is portable across multiple Java platforms?</th>
<th>Response Percent</th>
<th>Response Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>30.4%</td>
<td>96</td>
</tr>
<tr>
<td>No</td>
<td>69.6%</td>
<td>220</td>
</tr>
</tbody>
</table>
The Unsafe survey

7. If there was a "safe unsafe" standard (cross-platform) alternative for your use-cases (perhaps a new API, perhaps language changes, or both) would you be prepared to replace Unsafe with that alternative? If so under what conditions?

<table>
<thead>
<tr>
<th>Response</th>
<th>Percent</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes (perhaps with conditions, if so please state those conditions)</td>
<td>88.6%</td>
<td>280</td>
</tr>
<tr>
<td>No</td>
<td>11.4%</td>
<td>36</td>
</tr>
</tbody>
</table>
Why is Unsafe in the JDK?

• For low-level VM-based features
  • Often intrinsicsified by HotSpot
  • JNI is PITA!
• Not directly expressible in byte-code
• Should be the only point between “can’t do it” and “officially supported”
Functionality

- Peek and poke at memory both on and off heap
- Define, instantiate and check initialization of classes
- Park/unpark threads
- Memory fences
- Some other minor stuff...
Use-cases

• When you can peek and poke at memory the use-cases are many and varied

• Use-cases come down to two things
  • Performance; and/or
  • Work around JDK restrictions/limitations
Characterizing use-cases

- A number of vertical use-cases
- Each of which can be tackled independently
  - Some sooner than others
- Wean developers off Unsafe step-by-step
The Unsafe survey

<table>
<thead>
<tr>
<th>Reason</th>
<th>Response Percent</th>
<th>Response Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic access to fields and array elements (such as compare-and-swap)</td>
<td>44.1%</td>
<td>149</td>
</tr>
<tr>
<td>Off-heap memory operations (such as to emulate structures or packed objects)</td>
<td>63.6%</td>
<td>215</td>
</tr>
<tr>
<td>Deserialization hacks</td>
<td>36.4%</td>
<td>123</td>
</tr>
<tr>
<td>Fencing (to constrain re-ordering of memory operations)</td>
<td>22.5%</td>
<td>76</td>
</tr>
<tr>
<td>Access to private fields of another class</td>
<td>25.1%</td>
<td>85</td>
</tr>
<tr>
<td>Array access without bounds checks</td>
<td>32.5%</td>
<td>110</td>
</tr>
<tr>
<td>Other (please specify)</td>
<td>22.2%</td>
<td>75</td>
</tr>
</tbody>
</table>
Three general cases

- Off-heap
- Enhanced atomic access & fences
- De/Serialization
Four off-heap use-cases

- Reduce GC
- Efficient memory layout
- Very large collections
- Communicate across JVM boundary
## Use-case to Feature

<table>
<thead>
<tr>
<th>Use-case</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enhanced atomic access</td>
<td>JEP 193 Enhanced Volatiles</td>
</tr>
<tr>
<td>De/Serialization</td>
<td>JEP 187 Serialization 2.0</td>
</tr>
<tr>
<td>Reduce GC</td>
<td>Value types</td>
</tr>
<tr>
<td></td>
<td>JEP 189 Shenandoah: Low Pause GC</td>
</tr>
<tr>
<td>Efficient memory layout</td>
<td>Value types, Arrays 2.0 &amp; Layouts</td>
</tr>
<tr>
<td>Very Large Collections</td>
<td>Value types, Arrays 2.0 &amp; Layouts</td>
</tr>
<tr>
<td>Communicate across JVM boundary</td>
<td>Project Panama &amp; JEP 191 FFI</td>
</tr>
</tbody>
</table>
A Strategy

• Put features in Unsafe; use them in JDK
• Identify features that really belong in the Java programming model
• Add support for those features
• (Theoretically) garbage-collect them from Unsafe
Carrot and Stick

- Removing key functionality can hurt
- We need to make it easier for library/framework developers to migrate gradually
  - From release N with `Unsafe` to release N+1 without `Unsafe`
Enhanced Volatiles

- **Safe** enhanced atomic access to field and array elements

- Without the dynamic overhead of `Atomic*{Updater|Array}` classes

- Replace nearly all usages of `Unsafe` in `java.util.concurrent` classes
Success Metrics

• API should be at least as good as Unsafe

• Performance results should be close to Unsafe, and faster than Atomic* classes

• Good enough to replace Unsafe usages in java.util.concurrent classes

• Eat our own dog food updating classes in the JDK
Code in ForkJoinPool

```java
final ForkJoinTask<?> poll() {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (((a.length - 1) & b) << ASHIFT) + ABASE;
        t = (ForkJoinTask<?>)U.getObjectVolatile(a, j);
        if (t != null) {
            if (U.compareAndSwapObject(a, j, t, null)) {
                U.putOrderedInt(this, QBASE, b + 1);
                return t;
            }
        } else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```
Possible solution: MethodHandle

- A `MethodHandle` is a reference to an underlying method, constructor, or field
- Supports volatile access to a field
- Invocations inline surprisingly well
- With some tweaks can support other forms of access
```java
final ForkJoinTask<?> poll() {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (a.length - 1) & b;
        t = (ForkJoinTask<?>)ABASE_getVolatile.invokeExact(a, j);
        if (t != null) {
            if ((boolean) ABASE_compareAndSet.invokeExact(
                a, j, t, (ForkJoinTask) null)) {
                QBASE_setRelease.invokeExact(this, b + 1);
                return t;
            }
        } else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```
MethodHandle: Success Metrics

✗ API should be at least as good as Unsafe, preferably better

✓ Performance results should be close to Unsafe, and faster than Atomic* classes
Low Hanging Fruit

- The methods `monitorEnter/monitorExit/tryMonitorEnter` can be removed
- No one is using them
- Add a lexicographical byte comparator to `{Direct | Heap}ByteBuffer
- Cassandra, Guava, ...
In Safety...

• `sun.misc.Unsafe` is going away
• Accessible only from within the JDK
• Safe supported features planned that replace common unsafe use
• Preserving Java’s culture